Getting started with httr

httr quickstart guide

The goal of this document is to get you up and running with httr as quickly as possible. httr is designed to map closely to the underlying http protocol. I'll try and explain the basics in this intro, but I'd also recommend "HTTP: The Protocol Every Web Developer Must Know" or "HTTP made really easy".

This vignette (and parts of the httr API) derived from the excellent "Requests quickstart guide" by Kenneth Reitz. Requests is a python library similar in spirit to httr.

There are two important parts to http: the **request**, the data sent to the server, and the **response**, the data sent back from the server. In the first section, you'll learn about the basics of constructing a request and accessing the response. In the second and third sections, you'll dive into more details of each.

httr basics

To make a request, first load httr, then call `GET()` with a url:

```
library(httr)
r <- GET("http://httpbin.org/get")
```

This gives you a response object. Printing a response object gives you some useful information: the actual url used (after any redirects), the http status, the file (content) type, the size, and if it's a text file, the first few lines of output.

```
r
#> Response [http://httpbin.org/get]
#>   Date: 2020-07-20 14:19
#>   Status: 200
#>   Content-Type: application/json
#>   Size: 365 B
#> {
#>   "args": {},
#>   "headers": {
#>     "Accept": "application/json, text/xml, application/xml, */*",
#>     "Accept-Encoding": "deflate, gzip",
#>     "Host": "httpbin.org",
#>     "User-Agent": "libcurl/7.64.1 r-curl/4.3 httr/1.4.2",
#>     "X-Amzn-Trace-Id": "Root=1-5f15a7db-639798b6a3984d2402ca7808"
#>   },
#>   "origin": "73.115.118.38",
#> ...
```

You can pull out important parts of the response with various helper methods, or dig directly into the object:

```
status_code(r)
#> [1] 200
headers(r)
#> $date
#> [1] "Mon, 20 Jul 2020 14:19:07 GMT"
#>
#> $`content-type`
#> [1] "application/json"
#>
#> $`content-length`
#> [1] "365"
#>
#> $connection
#> [1] "keep-alive"
#>
#> $server
#> [1] "gunicorn/19.9.0"
#>
#> $`access-control-allow-origin`
#> [1] "*"
#>
#> $`access-control-allow-credentials`
#> [1] "true"
#>
#> attr(,"class")
#> [1] "insensitive" "list"
str(content(r))
#> List of 4
#>  $ args   : Named list()
#>  $ headers:List of 5
#>   ..$ Accept         : chr "application/json, text/xml, application/xml, */*"
#>   ..$ Accept-Encoding: chr "deflate, gzip"
#>   ..$ Host           : chr "httpbin.org"
#>   ..$ User-Agent     : chr "libcurl/7.64.1 r-curl/4.3 httr/1.4.2"
#>   ..$ X-Amzn-Trace-Id: chr "Root=1-5f15a7db-639798b6a3984d2402ca7808"
#>  $ origin : chr "73.115.118.38"
#>  $ url    : chr "http://httpbin.org/get"
```

I'll use `httpbin.org` throughout this introduction. It accepts many types of http request and returns json that describes the data that it received. This makes it easy to see what httr is doing.

As well as `GET()`, you can also use the `HEAD()`, `POST()`, `PATCH()`, `PUT()` and `DELETE()` verbs. You're probably most familiar with `GET()` and `POST()`: `GET()` is used by your browser when requesting a page, and `POST()` is (usually) used when submitting a form to a server. `PUT()`, `PATCH()` and `DELETE()` are used most often by web APIs.

The response

The data sent back from the server consists of three parts: the status line, the headers and the body. The most important part of the status line is the http status code: it tells you whether or not the request was successful. I'll show you how to access that data, then how to access the body and headers.

The status code

The status code is a three digit number that summarises whether or not the request was successful (as defined by the server that you're talking to). You can access the status code along with a descriptive

message using `http_status()`:

```
r <- GET("http://httpbin.org/get")
# Get an informative description:
http_status(r)
#> $category
#> [1] "Success"
#>
#> $reason
#> [1] "OK"
#>
#> $message
#> [1] "Success: (200) OK"

# Or just access the raw code:
r$status_code
#> [1] 200
```

A successful request always returns a status of 200. Common errors are 404 (file not found) and 403 (permission denied). If you're talking to web APIs you might also see 500, which is a generic failure code (and thus not very helpful). If you'd like to learn more, the most memorable guides are the http status cats.

You can automatically throw a warning or raise an error if a request did not succeed:

```
warn_for_status(r)
stop_for_status(r)
```

I highly recommend using one of these functions whenever you're using httr inside a function (i.e. not interactively) to make sure you find out about errors as soon as possible.

The body

There are three ways to access the body of the request, all using `content()`:

- `content(r, "text")` accesses the body as a character vector:

  ```
  r <- GET("http://httpbin.org/get")
  content(r, "text")
  #> No encoding supplied: defaulting to UTF-8.
  #> [1] "{\n  \"args\": {}, \n  \"headers\": {\n    \"Accept\": \"application/json, text/xml
  ```

  httr will automatically decode content from the server using the encoding supplied in the `content-type` HTTP header. Unfortunately you can't always trust what the server tells you, so you can override encoding if needed:

  ```
  content(r, "text", encoding = "ISO-8859-1")
  ```

  If you're having problems figuring out what the correct encoding should be, try `stringi::stri_enc_detect(content(r, "raw"))`.

- For non-text requests, you can access the body of the request as a raw vector:

  ```
  content(r, "raw")
  #>   [1] 7b 0a 20 20 22 61 72 67 73 22 3a 20 7b 7d 2c 20 0a 20 20 22 68 65 61 64 65
  #>  [26] 72 73 22 3a 20 7b 0a 20 20 20 20 22 41 63 63 65 70 74 22 3a 20 22 61 70 70
  #>  [51] 6c 69 63 61 74 69 6f 6e 2f 6a 73 6f 6e 2c 20 74 65 78 74 2f 78 6d 6c 2c 20
  #>  [76] 61 70 70 6c 69 63 61 74 69 6f 6e 2f 78 6d 6c 2c 20 2a 2f 2a 22 2c 20 0a 20
  #> [101] 20 20 20 22 41 63 63 65 70 74 2d 45 6e 63 6f 64 69 6e 67 22 3a 20 22 64 65
  #> [126] 66 6c 61 74 65 2c 20 67 7a 69 70 22 2c 20 0a 20 20 20 20 22 48 6f 73 74 22
  #> [151] 3a 20 22 68 74 74 70 62 69 6e 2e 6f 72 67 22 2c 20 0a 20 20 20 20 22 55 73
  #> [176] 65 72 2d 41 67 65 6e 74 22 3a 20 22 6c 69 62 63 75 72 6c 2f 37 2e 36 34 2e
  #> [201] 31 20 72 2d 63 75 72 6c 2f 34 2e 33 20 68 74 74 72 2f 31 2e 34 2e 32 22 2c
  #> [226] 20 0a 20 20 20 20 22 58 2d 41 6d 7a 6e 2d 54 72 61 63 65 2d 49 64 22 3a 20
  #> [251] 22 52 6f 6f 74 3d 31 2d 35 66 31 35 61 37 64 62 2d 61 37 31 33 39 66 66 30
  #> [276] 30 39 32 62 34 31 66 38 36 37 63 35 30 30 34 32 22 0a 20 20 7d 2c 20 0a 20
  #> [301] 20 22 6f 72 69 67 69 6e 22 3a 20 22 37 33 2e 31 31 35 2e 31 31 38 2e 33 38
  #> [326] 22 2c 20 0a 20 20 22 75 72 6c 22 3a 20 22 68 74 74 70 3a 2f 2f 68 74 74 70
  #> [351] 62 69 6e 2e 6f 72 67 2f 67 65 74 22 0a 7d 0a
  ```

  This is exactly the sequence of bytes that the web server sent, so this is the highest fidelity way of saving files to disk:

  ```
  bin <- content(r, "raw")
  writeBin(bin, "myfile.txt")
  ```

- httr provides a number of default parsers for common file types:

  ```
  # JSON automatically parsed into named list
  str(content(r, "parsed"))
  #> List of 4
  #>  $ args   : Named list()
  #>  $ headers:List of 5
  #>   ..$ Accept         : chr "application/json, text/xml, application/xml, */*"
  #>   ..$ Accept-Encoding: chr "deflate, gzip"
  #>   ..$ Host           : chr "httpbin.org"
  #>   ..$ User-Agent     : chr "libcurl/7.64.1 r-curl/4.3 httr/1.4.2"
  #>   ..$ X-Amzn-Trace-Id: chr "Root=1-5f15a7db-a7139ff0092b41f867c50042"
  #>  $ origin : chr "73.115.118.38"
  #>  $ url    : chr "http://httpbin.org/get"
  ```

  See `?content` for a complete list.

  These are convenient for interactive usage, but if you're writing an API wrapper, it's best to parse the text or raw content yourself and check it is as you expect. See the API wrappers vignette for more details.

The headers

Access response headers with `headers()`:

```
headers(r)
#> $date
#> [1] "Mon, 20 Jul 2020 14:19:07 GMT"
#>
```

```
#> $`content-type`
#> [1] "application/json"
#>
#> $`content-Length`
#> [1] "365"
#>
#> $connection
#> [1] "keep-alive"
#>
#> $server
#> [1] "gunicorn/19.9.0"
#>
#> $`access-control-allow-origin`
#> [1] "*"
#>
#> $`access-control-allow-credentials`
#> [1] "true"
#>
#> attr(,"class")
#> [1] "insensitive" "list"
```

This is basically a named list, but because http headers are case insensitive, indexing this object ignores case:

```
headers(r)$date
#> [1] "Mon, 20 Jul 2020 14:19:07 GMT"
headers(r)$DATE
#> [1] "Mon, 20 Jul 2020 14:19:07 GMT"
```

Cookies

You can access cookies in a similar way:

```
r <- GET("http://httpbin.org/cookies/set", query = list(a = 1))
cookies(r)
#>        domain  flag path secure expiration name value
#> 1 httpbin.org FALSE    /  FALSE      <NA>    a     1
```

Cookies are automatically persisted between requests to the same domain:

```
r <- GET("http://httpbin.org/cookies/set", query = list(b = 1))
cookies(r)
#>        domain  flag path secure expiration name value
#> 1 httpbin.org FALSE    /  FALSE      <NA>    a     1
#> 2 httpbin.org FALSE    /  FALSE      <NA>    b     1
```

The request

Like the response, the request consists of three pieces: a status line, headers and a body. The status line defines the http method (GET, POST, DELETE, etc) and the url. You can send additional data to the server in the url (with the query string), in the headers (including cookies) and in the body of `POST()`, `PUT()` and `PATCH()` requests.

The url query string

A common way of sending simple key-value pairs to the server is the query string: e.g. `http://httpbin.org/get?key=val`. httr allows you to provide these arguments as a named list with the `query` argument. For example, if you wanted to pass `key1=value1` and `key2=value2` to `http://httpbin.org/get` you could do:

```
r <- GET("http://httpbin.org/get",
  query = list(key1 = "value1", key2 = "value2")
)
content(r)$args
#> $key1
#> [1] "value1"
#>
#> $key2
#> [1] "value2"
```

Any `NULL` elements are automatically dropped from the list, and both keys and values are escaped automatically.

```
r <- GET("http://httpbin.org/get",
  query = list(key1 = "value 1", "key 2" = "value2", key2 = NULL))
content(r)$args
#> $`key 2`
#> [1] "value2"
#>
#> $key1
#> [1] "value 1"
```

Custom headers

You can add custom headers to a request with `add_headers()`:

```
r <- GET("http://httpbin.org/get", add_headers(Name = "Hadley"))
str(content(r)$headers)
#> List of 7
#>  $ Accept         : chr "application/json, text/xml, application/xml, */*"
#>  $ Accept-Encoding: chr "deflate, gzip"
#>  $ Cookie         : chr "b=1; a=1"
#>  $ Host           : chr "httpbin.org"
#>  $ Name           : chr "Hadley"
#>  $ User-Agent     : chr "libcurl/7.64.1 r-curl/4.3 httr/1.4.2"
#>  $ X-Amzn-Trace-Id: chr "Root=1-5f15a7db-9cc30cee85eb44aa2325b738"
```

(Note that `content(r)$header` retrieves the headers that httpbin received. `headers(r)` gives the headers that it sent back in its response.)

Cookies

Cookies are simple key-value pairs like the query string, but they persist across multiple requests in a session (because they're sent back and forth every time). To send your own cookies to the server, use `set_cookies()`:

```
r <- GET("http://httpbin.org/cookies", set_cookies("MeWant" = "cookies"))
content(r)$cookies
#> $MeWant
#> [1] "cookies"
#>
#> $a
#> [1] "1"
#>
#> $b
#> [1] "1"
```

Note that this response includes the `a` and `b` cookies that were added by the server earlier.

Request body

When `POST()`ing, you can include data in the `body` of the request. httr allows you to supply this in a number of different ways. The most common way is a named list:

```
r <- POST("http://httpbin.org/post", body = list(a = 1, b = 2, c = 3))
```

You can use the `encode` argument to determine how this data is sent to the server:

```
url <- "http://httpbin.org/post"
body <- list(a = 1, b = 2, c = 3)

# Form encoded
r <- POST(url, body = body, encode = "form")
# Multipart encoded
r <- POST(url, body = body, encode = "multipart")
# JSON encoded
r <- POST(url, body = body, encode = "json")
```

To see exactly what's being sent to the server, use `verbose()`. Unfortunately due to the way that `verbose()` works, knitr can't capture the messages, so you'll need to run these from an interactive console to see what's going on.

```
POST(url, body = body, encode = "multipart", verbose()) # the default
POST(url, body = body, encode = "form", verbose())
POST(url, body = body, encode = "json", verbose())
```

`PUT()` and `PATCH()` can also have request bodies, and they take arguments identically to `POST()`.

You can also send files off disk:

```
POST(url, body = upload_file("mypath.txt"))
POST(url, body = list(x = upload_file("mypath.txt")))
```

(`upload_file()` will guess the mime-type from the extension - using the `type` argument to override/supply yourself.)

These uploads stream the data to the server: the data will be loaded in R in chunks then sent to the remote server. This means that you can upload files that are larger than memory.

See `POST()` for more details on the other types of thing that you can send: no body, empty body, and character and raw vectors.

Built with

```
sessionInfo()
#> R version 3.6.3 (2020-02-29)
#> Platform: x86_64-apple-darwin15.6.0 (64-bit)
#> Running under: macOS Catalina 10.15.5
#>
#> Matrix products: default
#> BLAS:   /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRblas.0.dylib
#> LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib
#>
#> Locale:
#> [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> other attached packages:
#> [1] httr_1.4.2
#>
#> Loaded via a namespace (and not attached):
#>  [1] compiler_3.6.3       R6_2.4.1             magrittr_1.5
#>  [4] htmltools_0.4.0.9003 tools_3.6.3          curl_4.3
#>  [7] yaml_2.2.1           stringi_1.4.6        rmarkdown_2.3.1
#> [10] knitr_1.28           jsonlite_1.6.1       stringr_1.4.0
#> [13] digest_0.6.25        xfun_0.13            rlang_0.4.7
#> [16] evaluate_0.14
```